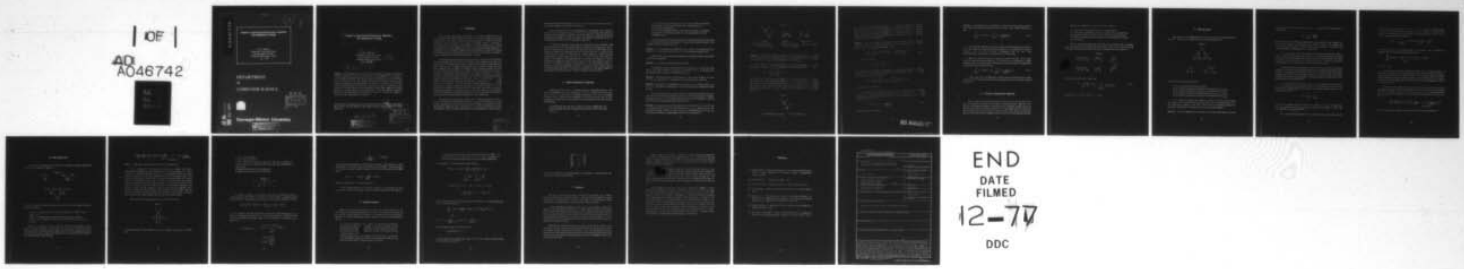
AD-A046 742 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 12/1
ANALYSIS OF ASYNCHRONOUS MULTIPROCESSOR ALGORITHMS WITH APPLICA--ETC(U)
JUL 77 J T ROBINSON N00014-76-C-0370

UNCLASSIFIED NL



END
DATE FILMED
12-77
DDC

(12) 9

Analysis of Asynchronous Multiprocessor Algorithms with Applications to Sorting

John T. Robinson
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
July 1977

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Analysis of Asynchronous Multiprocessor Algorithms with Applications to Sorting

John T. Robinson
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

July 1977

21 p.

Interim rept.,

Abstract - Efficient algorithms for asynchronous multiprocessor systems must achieve a balance between low process communication and high adaptability to variations in process speed. Algorithms which employ problem decomposition can be classified as static and dynamic. Static and dynamic algorithms are particularly suited for low process communication and high adaptability, respectively. In order to find the "best" method, something about mean execution times must be known. Techniques for the analysis of the mean execution time are developed for each type of algorithm, including applications of order statistics and queueing theory. These techniques are applied in detail to (1) static generalizations of quicksort, (2) static generalizations of merge sort, and (3) a dynamic generalization of quicksort.

This research was supported in part by the National Science Foundation under grant MCS75-222-55 and the Office of Naval Research under contract N00014-76-C-0370, NR044-422.

NSF-MCS-75-22255

1 - Introduction

We consider the design and analysis of k-process algorithms for an asynchronous multiprocessor system, which consists of k or more processors sharing a common memory by means of a switch or connecting network. In addition there is an operating system providing such functions as process creation, scheduling of processes, allocation of memory, synchronization, etc. A real example of such a system is described in [7], and a general discussion of asynchronous parallel algorithms is presented in [5]. A k-process algorithm will be presented by giving the procedure each process executes when assigned a processor. We will assume that a processor is always available for any of the k processes that is runnable.

Given a task we wish to execute on such a system, in order to exploit parallelism we must decompose the task into a set of subtasks. Some subtasks cannot begin until others which they depend upon finish; this establishes a precedence relation between tasks. Inefficiency in an algorithm arises when some process must spend too much *time waiting for other processes to complete* subtasks, and again towards the end of execution when there are fewer than k subtasks. Attempts to remedy this by "evenly" dividing the original task are hopeless, since task execution time will vary due to variations in the input, the effects of other users, properties of the operating system, processor-memory interference, and many other causes. Any efficient al[illegible] must adapt to these variations. However, this adaptation is expensive, in t[illegible]quires process communication. Thus the trade-off between adaptability a[illegible] process communication must be considered in the design of multiprocessor algorithms. In the algorithms considered in this paper, process communication takes place by means of global data accessible by all processes. Since in many cases access to this global data must be confined to a critical section, one cause of process communication overhead is the interference between processes seeking access to this global data.

Two methods of decomposition naturally arise: (1) static decomposition, in which the set of subtasks and their precedence relations are known before execution, and (2) dynamic decomposition, in which the set of subtasks changes during execution. Static decomposition algorithms offer the possibility of very low process communication, providing there are not too many tasks; however, their adaptability is limited. Dynamic

decomposition algorithms can adapt to variations in task execution time very well, but only at the expense of high process communication.

Given a problem which can be decomposed into subproblems, which method is best? Is the extra expense necessary for fast process communication (thus supporting efficient dynamic algorithms) justified? If a dynamic algorithm is used, how far should decomposition proceed? In order to answer these questions we need techniques for finding mean execution times for these types of algorithms.

In section 2 algorithms employing static decomposition are considered. We develop techniques for finding the probability distribution of total execution time in terms of the distributions of individual task execution times, and when these are not known, techniques for finding bounds on the mean execution time. In section 3, the mean execution time for a simple model of a dynamic algorithm is found, assuming exponentially distributed task execution times. In sections 4 and 5 the results of section 2 are applied to static generalizations of quicksort and merge sort. Certain partitioning strategies are shown to be unsuitable for a static decomposition version of quicksort. In addition, a parallel merging algorithm is presented and analyzed. In section 6 a dynamic generalization of quicksort is presented. Using a result of section 3, the mean execution time is found, and an expression for the optimal degree of decomposition is derived. Section 7 contains a summary of the main results.

2 - Static Decomposition Algorithms

Given a set of tasks $T_1, T_2, \ldots T_n$ partially ordered by a precedence relation $<$, we call T_i a predecessor of T_j (T_j a successor of T_i) if $T_i < T_j$. If there is no task U such that $T_i < U < T_j$, T_i is said to be an immediate predecessor of T_j (T_j an immediate successor of T_i). Tasks with no predecessors are called initial, and tasks with no successors are called final. In the execution of the static algorithm, each process does the following:

(1) Select either an initial task or a task all of whose predecessors have been completed, which has not already been selected. Check in the order $T_1, T_2, \ldots T_n$.

(2) If no task can be selected, go to sleep, unless all tasks have already been selected, in which case terminate. When awakened go to (1).
(3) Execute the selected task.
(4) For each immediate successor of the task, record that an immediate predecessor has completed, and wake up a sleeping process if possible.
(5) Repeat from (1).

For the purposes of analysis we assume that steps (1),(2),(4), and (5) take zero time, and that the execution time of task T_i is given by the random variable t_i, with cumulative distribution function (c.d.f.) F_i.

Definition - The task-graph G associated with $T_1,T_2,\ldots T_n$ and $<$ is a directed graph with nodes $T_1,T_2,\ldots T_n$ and arrows from T_i to T_j if T_i is an immediate predecessor of T_j.

Note that there is a one-to-one correspondence between partially ordered sets of tasks and task-graphs.

Definition - G is a chain if the tasks are totally ordered.

The length of a chain is the number of tasks in the chain. If in a chain the initial task is T_i and the final task is T_j we say it is a chain from T_i to T_j. A sub-graph of a task-graph G which is a chain is said to be a chain in G.

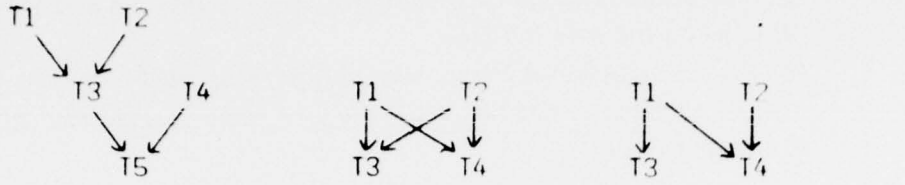
Definition - The level of a task T in a task-graph G is the maximum length of any chain in G from an initial task to T. The depth of G is the maximum level of any task.

Definition - A set of tasks is independent if for any tasks T_i, T_j in the set, neither $T_i<T_j$ nor $T_j<T_i$. The width of a task-graph is the maximum size of any independent subset of tasks.

Given a task-graph G, let t_G be the random variable representing total execution time (the time from when all processes are started until the last process terminates). Assume t_G has c.d.f. F_G. In the following definition a class of task-graphs is defined for which F_G can be expressed simply in terms of the F_i.

Definition - Let $C_1,C_2,\ldots C_m$ be all chains from initial to final tasks in G. For each chain C_i containing tasks $T_{i_1},T_{i_2},\ldots$, let E_i be the expression $(x_{i_1}\cdot x_{i_2}\cdot\ldots)$, where $x_1,x_2,\ldots x_n$ are polynomial variables. Then G is said to be simple if the polynomial $E_1+E_2+\ldots+E_m$ can be factored so that each variable appears exactly once (see figure 2.1).

Figure 2.1



Simple

$x_1x_3x_5+x_2x_3x_5+x_4x_5$
$=((x_1+x_2)x_3+x_4)x_5$

Simple

$x_1x_3+x_1x_4+x_2x_3+x_2x_4$
$=(x_1+x_2)(x_3+x_4)$

Non-simple

$x_1x_3+x_1x_4+x_2x_4$

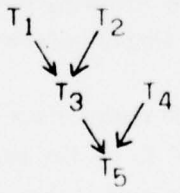
Theorem - If k≥width(G), then t_G can be expressed in terms of the t_i using only + and max. Furthermore, if G is simple and the t_i are independent, then F_G can be expressed in terms of the F_i using only · (multiplication) and * (convolution).

Proof: Note that since k≥width(G) each task begins immediately after its last predecessor completes. Let $C_1,C_2,\ldots C_m$ be all chains from initial to final tasks. Then

$$t_G = \max_{1\le i\le m} \left(\sum_{T_j \in C_i} t_j \right) .$$

Next note that + and max are commutative and associative operations, and that + distributes over max (i.e., max(a,b)+c=max(a+c,b+c)). Thus if G is simple the expression for t_G above can be factored in terms of max and + so that each random variable appears only once. Then, if the t_i are independent, the expression for F_G may be found by substituting F_i for t_i, * for +, and · for max in the expression for t_G (see figure 2.2).

Figure 2.2



$t_G=\max(\max(t_1,t_2)+t_3,t_4)+t_5$ $\qquad$ $F_G=(((F_1F_2)*F_3)F_4)*F_5$

Thus in the proof of this theorem we have a method for calculating the c.d.f. of total execution time for simple task-graphs with independent task execution times, providing we know the c.d.f. of the execution time of each task. When the c.d.f.s of each task's execution time are not known, the best we can do is derive bounds on mean execution time, such as those of the following theorem. The expected value of a random variable x is denoted by E(x).

Theorem - Given a task-graph G with k≥width(G) and with the t_i independent, let $C_1, C_2, \ldots C_m$ be all chains in G from initial to final tasks. Also let H_i be the set of all tasks of level i, for $1 \le i \le l$ where l=depth(G). Then

$$\max_{1 \le i \le m} \left(\sum_{T_j \in C_i} E(t_j) \right) \le E(t_G) \le \sum_{1 \le i \le l} E\left(\max_{T_j \in H_i} t_j \right) \qquad (2.1)$$

Proof: From above,

$$t_G = \max_{1 \le i \le m} \left(\sum_{T_j \in C_i} t_j \right) .$$

The lower bound then follows from $E(\max\{x_i\}) \ge \max\{E(x_i)\}$ for any random variables x_i. For the upper bound, let $t_0 \equiv 0$ and define f(i,j)=0 if $C_i \cap H_j$ is empty, otherwise f(i,j) is the index of the single task in $C_i \cap H_j$. Then

$$t_G = \max_{1 \le i \le m} \left(\sum_{1 \le j \le l} t_{f(i,j)} \right) \le \sum_{1 \le j \le l} \left(\max_{1 \le i \le m} (t_{f(i,j)}) \right)$$

from which the result follows.

The upper bound in equation 2.1 is useful only if something can be said about $E(\max\{t_j\})$. An applicable result from order statistics (see [2]) is that if the independent random variables $x_1, x_2, \ldots x_m$ are identically distributed with mean u and standard deviation s, then

$$E(\max\{x_i\}) \le u + \frac{m-1}{\sqrt{2m-1}} s \qquad (2.2)$$

Hence the following corollary:

Corollary - If k≥width(G), the t_i are independent, depth(G)=l, and the m_j tasks on level j have identically distributed execution times with mean u_j and standard deviation s_j, then

$$\sum_{1\le j\le l} u_j \le E(t_G) \le \sum_{1\le j\le l} \left(u_j + \frac{m_j - 1}{\sqrt{2m_j - 1}} s_j \right) \qquad (2.3)$$

Let w=width(G). When w>k, F_G cannot in general be expressed simply in terms of the F_i, even when G is simple and the t_i are independent. For example, let G consist of T_1, T_2, T_3 with the set $\{T_1, T_2, T_3\}$ independent, and let k=2. Then $t_G = \max(\min(t_1, t_2) + t_3, \max(t_1, t_2))$, and t_G cannot be simplified further.

When w>k, the lower bounds for $E(t_G)$ given above still hold. For an upper bound we take the following approach. It is assumed that w processes are created, and each process has a processor available at least k/w of the time. For example, the bound given in the corollary becomes

$$\sum_{1\le j\le l} u_j \le E(t_G) \le \frac{w}{k} \sum_{1\le j\le l} \left(u_j + \frac{m_j - 1}{\sqrt{2m_j - 1}} s_j \right) \qquad (2.4)$$

Finally, when the t_i are dependent, in general special techniques must be used, such as those in the analysis of partitioning strategies (section 4) or parallel merging (section 5).

3 - A Dynamic Decomposition Algorithm

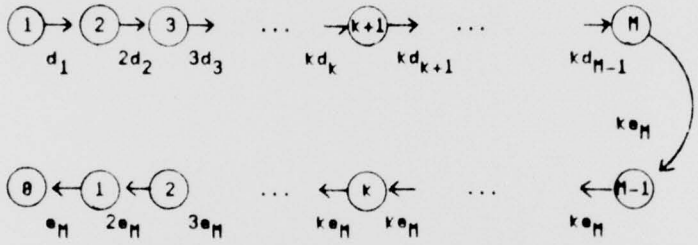
Given a task T and a procedure which decomposes a task into two tasks which may be executed concurrently, we consider the following dynamic algorithm: First, there is a decomposition phase, in which each process repeatedly removes tasks from the task-queue TQ (which initially contains only T), decomposes the task and inserts the two new tasks in TQ, until there is a total of M tasks. Next, there is an execution phase, in which each process repeatedly removes tasks from TQ and executes the task.

We analyze this algorithm under the following assumptions:

(1) In this section the time to access TQ is assumed to be 0.
(2) The time to decompose a task is assumed to be exponentially distributed with mean d_i^{-1}, where i is the current total number of tasks.
(3) The time to execute a task is assumed to be exponentially distributed with mean e_M^{-1}.

We use standard queueing theory techniques in the analysis (see for example [3]). Adopting as a state variable the total number of tasks in TQ or currently being executed or decomposed, the state-transition-rate diagram is given by figure 3.1.

Figure 3.1



The mean execution time is found to be:

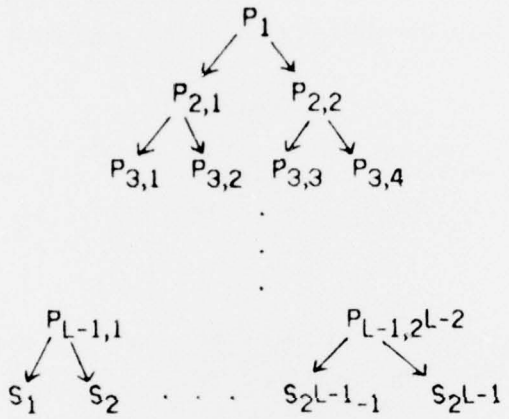
$$T = \frac{1}{e_M}\left(\frac{M-1}{k} + H_k\right) + \sum_{1 \le i \le M-1} \frac{1}{\min(i,k)d_i} \qquad (3.1)$$

where $H_k = (1 + 1/2 + 1/3 + \ldots + 1/k)$.

4 - Static Quicksort

We consider a static generalization of quicksort as given by the task-graph of figure 4.1 (see [6] for a complete discussion of sequential quicksort):

Figure 4.1



The tasks may be described as follows:

(1) P_1 is a partition of the file to be sorted.
(2) $P_{i,j}$ (j odd) is a partition of the left subfile produced by $P_{i-1,(j+1)/2}$.
(3) $P_{i,j}$ (j even) is a partition of the right subfile produced by $P_{i-1,j/2}$.
(4) S_j (j odd) is a quicksort of the left subfile produced by $P_{L-1,(j+1)/2}$.
(5) S_j (j even) is a quicksort of the right subfile produced by $P_{L-1,j/2}$.

First consider the simplest case, where k is a power of 2 and $L=1+\lg(k)$ (where lg is $\log_2$). In this case the width of the task graph is k. The question arises as to what partitioning strategy to use, that is, how should the partitioning element be selected in the P tasks? First a definition of asysmptotic mean speedup:

Definition - Given an algorithm for k processes, let the mean total execution time be

$T_k(N)$, where N is the size of the input. Then the asymptotic mean speedup S_k is defined to be

$$S_k = \lim_{N\to\infty} \frac{T_1(N)}{T_k(N)}$$

We would prefer a partitioning strategy which gives asymptotic mean speedup of k even in the simplest case; strategies which depend on large L for speedup are unsuitable since the number of tasks increases exponentially with L, and one of the main advantages of static algorithms is low overhead.

It is now necessary to make some assumptions about the execution times of tasks. In the sequential analysis of quicksort it is found that partitioning a file of size N takes O(N) time with standard deviation O(N), and that sorting a file of size N takes O(N lg(N)) time with standard deviation O(N) (see [6]). Thus in analyzing asymptotic mean speedup it is only necessary to consider the sorting task times.

(1) When the partitioning element for a partition of a file of size M is selected at random, it is natural to assume that either subfile size is uniformly distributed between 0 and M. This, together with the fact that the sum of the subfile sizes is M, gives an expected maximum subfile size of 3M/4. Using this, it is easy to show that of the k subfiles to be sorted in the sorting tasks, the expected maximum subfile size is at least $(3/4)^{\lg(k)}N$, which implies $S_k \le k^{\lg(4/3)}$.

(2) If the median of three method is used to select the partitioning element, and if it is assumed that the final position of each of the three elements in the subfile is uniformly distributed between 0 and M, then the probability density function for the size of either subfile is:

$$f(x) = \frac{6}{M}\left(1 - \frac{x}{M}\right)\frac{x}{M}$$

This gives an expected maximum subfile size of 11M/16. As in (1), it can be shown that the expected maximum size of the subfiles to be sorted is larger than $(11/16)^{\lg(k)}N$. It follows $S_k \le k^{\lg(16/11)}$.

(3) If the partitioning elements for all partitioning tasks are found using the

method of samplesort (first pick k-1 elements randomly, sort, and use these for the k-1 P tasks), and if the final position of each of the k-1 elements is assumed to be uniformly distributed between 0 and N, then the probability density function for the size of the largest subfile to be sorted is:

$$f(x) = \sum_{1 \le j \le \lfloor N/x \rfloor} (-1)^{j-1} k(k-1) \binom{k-1}{j-1} \left(1 - \frac{jx}{N}\right)^{k-2} .$$

(See the discussion on the random division of an interval in [2]). It follows the expected maximum size of the subfiles to be sorted is:

$$\int_0^N x f(x)\,dx = \frac{N}{k} \sum_{1 \le j \le k} (-1)^{j-1} \binom{k}{j-1} \frac{1}{j} = \frac{H_k}{k} N.$$

Hence $S_k = k/H_k$.

(4) Finally we turn to the partitioning strategy of first finding the median (in O(M) time, where M is the size of the subfile) in each P task, and using the median as the partitioning element. This does give $S_k=k$, but it should be noted that median finding represents a large overhead. Unless process communication is extremely expensive, a dynamic generalization of quicksort (such as the one presented in section 6) is probably better.

If the mean and standard deviation of the time to quicksort a file of size M are $a_q M \lg(M)$ and $b_q M$, and the mean and standard deviation of the time to find the median of a file of size M and partition the file using the median as partitioning element are $a_p M$ and $b_p M$, then from equation 2.3 we find that the mean total execution time is less than

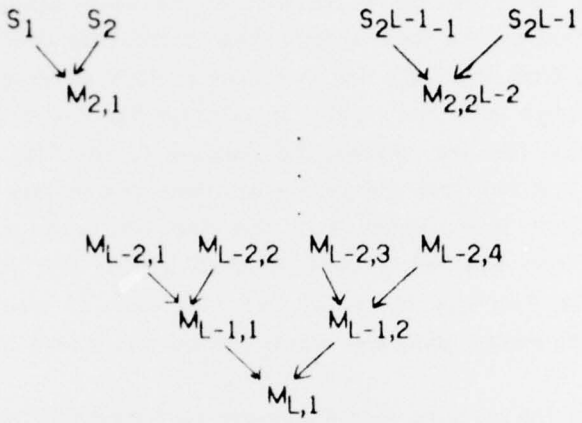
$$a_q \left(\frac{N}{k}\right) \lg\left(\frac{N}{k}\right) + \left[2a_p \left(1 - \frac{1}{k}\right) + \frac{k-1}{k} \frac{b_q}{\sqrt{2k-1}} + \sum_{1 \le j \le \lg(k)-1} \frac{2^{j-1}}{2^j} \frac{b_p}{\sqrt{2^{j+1}-1}} \right] \cdot N .$$

When L is greater than 1+lg(k) a similar result may be found using equation 2.4.

5 - Static Merge Sort

Consider a static generalization of merge sort as given by figure 5.1 (see [4] for a discussion of sequential merge sort):

Figure 5.1



The tasks may be described as follows, assuming the file to be sorted consists of records 1 through N:

(1) S_i is a merge sort of all the records between $(i-1)(N/2^{L-1})$ and $i(N/2^{L-1})+1$.
(2) $M_{2,i}$ is a merge of the two sorted files produced by S_{2i-1} and S_{2i}.
(3) $M_{i,j}$ $(i>2)$ is a merge of the two sorted files produced by $M_{i-1,2j-1}$ and $M_{i-1,2j}$.

When k is a power of 2 and $L=1+\lg(k)$, the width of the task graph is k and equation 2.3 may be applied. Assuming the time to merge sort a file of size N has mean $a_s N \lg(N)$ and standard deviation $b_s\sqrt{N}$, and that the time to merge two files of sizes M and N has mean $a_m(M+N)$ and standard deviation b_m (see [4]), we find that the mean total execution time is less than

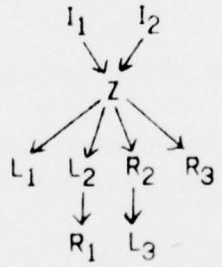
$$a_s\left(\frac{N}{k}\right)\lg\left(\frac{N}{k}\right)+2a_m\left(1-\frac{1}{k}\right)N+(k-1)\frac{b_s\sqrt{N}}{\sqrt{2k^2-k}}+\sum_{1\le j\le \lg(k)-1}(2^j-1)\frac{b_m}{\sqrt{2^{j+1}-1}}\ .$$

When L is larger than 1+lg(k) a similar result holds, using equation 2.4.

In the remainder of this section we consider one possible improvement: replacing the merging tasks with parallel merges. A two task merge of two files is possible by letting each task be an instance of the usual sequential two-way merge (see [4]), except that in one task merging begins with the two smallest items of the two files (a merge from the left), and in the other task merging begins with the two largest items (a merge from the right). In addition the two tasks are interlinked as follows: in sequential two-way merge, the pointers to the files are compared to the ends of the files; in a two task merge, the pointers of one task are compared to the pointers of the other task. Because of this, the two tasks finish together almost exactly, providing one has not already finished before the other starts. We now assume a sequential two-way merge of two files each of size N takes time $2a_mN$. Hence a two process merge using the above method would take time a_mN.

Next consider the merging algorithm given by figure 5.2, for k=4:

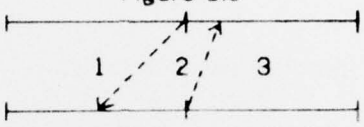
Figure 5.2



Assume the elements to be merged are $x_1<x_2<x_3<\ldots<x_N$ and $y_1<y_2<y_3<\ldots y_N$. The tasks are:

I_1: Insert $x_{\lfloor N/2 \rfloor}$ into the y_i's.
I_2: Insert $y_{\lfloor N/2 \rfloor}$ into the x_i's.
Z: The results of the insertions determine three pairs of subfiles, as shown in figure 5.3. Z determines the subfile pairs and initializes the L_i and R_i tasks.
L_i: Merge from the left of the i'th subfile pair.
R_i: Merge from the right of the i'th subfile pair.

Figure 5.3



If process 1 executes L_1 and process 2 executes L_2 and then R_1, process 1 finishes before or with process 2. Let the sizes of the subfiles in the second subfile pair be X and Y. The execution time for process 2, starting at the completion of Z, is:

$$a_m \max\left(\frac{X+Y}{2}\ ,\ \frac{X+Y}{2}+\left(N-Y\ -\ \frac{X+Y}{2}\right)\Big/2\right) \le a_m\left(\frac{N}{2}+\frac{|X-Y|}{4}\right)$$

since $(X+Y)/2 \le N/2$. The same result holds for the process executing R_2 and L_3. In order to find the distribution of $|X-Y|$, it is assumed all elements x_i, y_i are distinct, and that all permutations are equally likely. Then the probability of inserting $x_{\alpha N}$ in position i is:

$$P(y_i<x_{\alpha N}<y_{i+1}) = \frac{\binom{i+\alpha N-1}{\alpha N-1}\binom{N(2-\alpha)-i}{(1-\alpha)N}}{\binom{2N}{N}}$$

$$= \frac{\alpha N\ \binom{N}{i}\binom{N}{\alpha N}}{(i+\alpha N)\ \binom{2N}{i+\alpha N}}$$

$$\doteq \frac{2\alpha N}{(i+\alpha N)\sqrt{N\pi}}\, e^{-(i-\alpha N)^2/N}$$

using the normal approximation to the binomial distribution. This distribution is again approximately normal, with mean αN and standard deviation $\sqrt{N/2}$. Assuming X and Y are actually distributed normally, the mean of $|X-Y|$ can be calculated to be $\sqrt{2N/\pi}$. Hence,

$$E(t_G) \quad \leq \quad a_m\left(\frac{N}{k} + \sqrt{\frac{N}{8\pi}}\right) + O(\lg(N))$$

where the $O(\lg(N))$ term is from the insertion tasks.

Other merging algorithms for k=4 and for higher k can be devised by using various element insertion strategies. Similar techniques may be used in their analysis.

6 - Dynamic Quicksort

We may use the dynamic algorithm of section 3 for sorting, where tasks are considered to be subfiles, the decomposition of a task is a partition of the subfile into two subfiles, and the execution of a task is a sort of the subfile. In analyzing this algorithm we make the following assumptions, where the file to be sorted contains N records:

(1) If M is the total number of subfiles to be produced during the decomposition stage, the total number of task-queue accesses is 3M-2, and each process makes approximate average of 3M/k such accesses. We therefore assume the overhead due to process communication is linear in M, and is given by w(i

(2) When there are i subfiles, the mean subfile size is N/i. It is assumed the time needed to partition a subfile is exponentially distributed, and that when there is a total of i subfiles the mean time is aN/i.

(3) During the task execution phase, the average subfile size is N/M. It is assumed the time to sort one of the M subfiles produced by decompositioning is exponentially distributed, with mean b(N/M)ln(N/M).

From equation 3.1, the mean execution time T(M,N,k) is:

$$T(M,N,k) = w(k)M + b\left(\frac{N}{M}\right)\ln\left(\frac{N}{M}\right)\left(\frac{M-k}{k} + H_k\right) +$$

$$\sum_{1\le i\le k-1} a\frac{N}{i^2} + \sum_{k\le i\le M-1} a\frac{N}{ki}$$

$$= w(k)M + \frac{N}{k}\left(b \ln N - aH_{k-1} + aH_{M-1} - b \ln M\right)$$

$$+ b\left(\frac{N}{M}\right)\ln\left(\frac{N}{M}\right)(H_k - 1) + aNH_{k-1} \quad (2)$$

Given N and k, we seek to find M so as to minimize T(M,N,k). If we approximate H_{M-1} by ln(M), then M must satisfy

$$\frac{\partial T}{\partial M} = w(k) + \frac{N(a-b)}{kM} + bN(H_k-1)\left(\frac{\ln M - \ln N - 1}{M^2}\right)$$

$$= 0 .$$

Let $A = \dfrac{w(k)}{bN(H_k - 1)}$ and $B = \dfrac{(a-b)}{bk(H_k-1)}$,

then the optimal value of M is the solution of

$$M \cdot e^{(AM^2+BM-1)} = N.$$

A short table of the optimal integer value of M for various values of w(k)/b follows, for the case k=4, a=b, $N=10^6$:

$\mu(4)/b$	M
10	930
10^2	313
10^3	105
10^4	35
10^5	11

Thus, given a,b,N, and k, the optimal degree of decomposition is determined by w(k), the process communication overhead.

7 - Summary

We have classified asynchronous multiprocessor algorithms which employ problem decomposition as static and dynamic. Static decomposition algorithms require little process communication and would be well-suited for systems where process communication is expensive, e.g., "loosely-coupled" computer networks.

A static decomposition algorithm is described by a task-graph. Simple task-graphs have the property that there is a simple expression for the probability distribution of total execution time in terms of the probability distributions of each task, providing the result of one task does not affect the execution time of another. If the probability distributions of each task's execution time are unknown, it is still possible to bound mean total execution times providing the means and variances of task execution times are known.

Regarding the upper bound given by equation 2.3, the bound is tight in that task-graphs and task execution time probability distributions may be constructed so that equality holds, using distributions derived in [2]. Any improved bound would require either more detailed information about the partial ordering of the tasks in the expression of the bound, or additional assumptions about the probability distributions of task execution times.

When process communication is inexpensive, dynamic decomposition algorithms are suitable. One technique for analyzing these algorithms is by means of a queueing model. Queueing models may be used in analyzing other types of asynchronous parallel algorithms as well (e.g., in [1] a queueing model is used to analyze asynchronous iterative methods).

For [illegible] decomposition algorithms the bounds derived in section 2 may be directly [illegible] as static quicksort with median finding and static merge sort. In other cases [illegible] task execution times are dependent other techniques must be used. This is the case for static quicksort when median finding is not used and in the parallel merging algorithm presented. These algorithms have dependent task execution times since there are tasks where the input size depends on the result of a previous task.

The assumption that process communication overhead is negligible in static decomposition algorithms is valid only if the total number of tasks is not very large. For this reason we have given bounds on mean execution time only for those algorithms in which the width of the task-graph is k (although a technique for greater width task-graphs has also been presented). These bounds give an indication of the performance that can be expected when process communication overhead is high enough to warrant the use of static decomposition. However, in dynamic decomposition algorithms we may choose the degree of decomposition, which should ideally be chosen so as to balance process communcication overhead and adaptability to variations in the execution times of tasks. For example, by applying a queueing model to a dynamic generalization of quicksort, we have derived an expression relating process communication overhead and the optimal degree of decomposition.

References

[1] Baudet, Gerard "Numerical Computation on Asynchronous Multiprocessors", Thesis Proposal, Department of Computer Science, Carnegie-Mellon University, 1976

[2] David, Herbert A. *Order Statistics*, Wiley, 1970

[3] Kleinrock, Leonard *Queueing Systems*, vol. 1, Wiley-Interscience, 1975

[4] Knuth, Donald *The Art of Computer Programming*, vol. 3, Addison-Wesley, 1972

[5] Kung, H. T. "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", *Algorithms and Complexity - New Directions and Recent Results*, ed. J. F. Traub, pp. 153-200, Academic Press, 1976

[6] Sedgewick, Robert *Quicksort*, Ph.D. Thesis, Computer Science Department, Stanford University, 1975

[7] Wulf, W. A., and C.G. Bell "C.mmp - A Multi-Mini-Processor", *Proceedings of the AFIPS 1972 Fall Joint Computer Conference*, vol. 41, pp. 765-777, 1972

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ANALYSIS OF ASYNCHRONOUS MULTIPROCESSOR ALGORITHMS WITH APPLICATIONS TO SORTING		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John T. Robinson		8. CONTRACT OR GRANT NUMBER(s) MCS75-222-55 N00014-76-C-0370; NR 044-422
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE July 1977 13. NUMBER OF PAGES 20
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Efficient algorithms for asynchronous multiprocessor systems must achieve a balance between low process communication and high adaptability to variations in process speed. Algorithms which employ problem decomposition can be classified as static and dynamic. Static and dynamic algorithms are particularly suited for low process communication and high adaptability, respectively. In order to find the "best" method, something about mean execution times must be known. Techniques for the analysis of the mean execution time are developed for each type of algorithm, including applications of order statistics and queueing theory. These techniques are applied in detail to (1) static generalizations of quicksort, (2) static generalizations of merge sort, and (3) a dynamic generalization of quicksort.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)